

SANDIA REPORT

SAND88—3198 • UC—32

Unlimited Release

Printed March 1989

Security Policy Concepts for Microprocessor-Based Systems

Robert M. Axline, Jr., Richard C. Ormesher

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors or subcontractors.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Distribution
Category UC-32

SAND88-3198
Unlimited Release
Printed March 1989

**Security Policy Concepts for
Microprocessor-Based Systems**

Robert M. Axline, Jr.
Safety Assessment Technology Division

and

Richard C. Ormesher
Space Systems Division

Sandia National Laboratories
Albuquerque, NM 87185

ABSTRACT

This report presents security policies for microprocessor-based systems and gives an example of how to enforce these policies, using an independent, hardware-based monitor, in a hypothetical single-processor system. The purpose of these policies is to detect erroneous behavior of the microprocessor system and to guarantee that accesses (read, write, or execute), by executable procedures, to the various system resources (other procedures, data areas, and peripheral ports) are in accordance with rules that are defined precisely and completely. We present the main result of our research as a "Second-Order Security Policy", which describes a segmentation of system resources into a number of "Blocks" and defines access rights of each "Process Block" to all Blocks in the system. The hardware-monitor example is a conceptual design of an independent monitor that we believe can be built to enforce the second-order policy in real time. This approach will be effective in preventing erroneous accesses to data structures and peripherals and in detecting errors in the transfer of program control from Block to Block.

Acknowledgements

We want to thank those who reviewed this report and those who attended our recent briefings on this subject. These people provided helpful comments on how to improve this description of our work. Special thanks go to Lori Parrott and Terry Axline, who contributed their technical editing skills, and to Jim Zipay, Blase Gaude, and Jim Gosler, whose suggestions helped make this report more technically accurate and complete.

We also want to acknowledge the work of Masood Namjoo and Edward J. McCluskey that is described in reference [8]. We discovered that reference after we had completed the first draft of this report. Their work, performed in 1982, contains many of the elements of the security policy concepts we independently derived in the course of our research. Later in this report, we will briefly describe their approach and contrast their work with ours.

Contents

Definitions	6
Introduction	9
Scope and Intent	10
Assumptions	10
Classes of Errors and Their Sources	11
Motivation for the Approach	12
A First-Order Security Policy	13
A Second-Order Security Policy	14
Implementation Methods Possible	20
A Conceptual Design Example	22
Interpretation of the Approach's Significance	33
Extensions to the Basic Approach	36
Some Problems With the Approach	39
Conclusions	41
References	42

Illustrations

<u>Figure</u>	<u>Title</u>	<u>Page</u>
1	Memory Segmentation; First-Order Policy	13
2	Resource Segmentation; Second-Order Policy	15
3	Rights to Execute an Object	17
4	Rights to Read or Write an Object	17
5	Hypothetical System with Monitor	23
6	External Hardware Monitor	24
7	Block Discriminator Bank	26
8	Block Discriminator Cell	27
9	Rights Vector Decoder	28
10	Rights Vector Format	29
11	Authenticator	31

<u>Table</u>		
1	First-Order Security Policy	14
2	Bus Content Versus Control Lines	16
3	Second-Order Policy: A Priori Rights Possible	18

Definitions

A--(from the conceptual example) the "auxiliary" bit of the Rights Vector; a value of one indicates that the auxiliary operation indicated by the AR control line is allowed for the current Subject/Object coincidence.

Access to Data Memory (ADM)--an access, by a Subject, that will result in data being read/written to/from a memory cell or port location.

Access to Instruction Memory (AIM)--an access, by a Subject, that will result in a new instruction being fetched and executed.

Address--a digital pattern appearing on the system's address bus. This pattern indicates which memory or port location is currently being accessed by the microprocessor.

Auxiliary Request (AR) Control Line--a control line indicating, when asserted, that an auxiliary operation, related in some known way to the current Subject/Object coincidence, is occurring.

B--the number of Blocks in the hypothetical system described in the conceptual design example in this report.

Block--A group of contiguous addresses, either of ROM or RAM memory, or of port locations. Blocks are distinct; that is, they don't have any common addresses.

Data Block--a Memory Block that can be read or written as data but cannot be executed.

DMA--direct memory access, an operation wherein a smart peripheral, in concert with a DMA controller, can bypass the microprocessor and write to or read from memory directly.

Executing--a Process Block is said to be executing when the IP points to an address within that Process Block.

Instruction Fetch (IF) Control Line--a control line indicating, when asserted, that the current access is an AIM.

Instruction Pointer (IP)--a pointer whose value equals the address of the instruction to be executed by the processor.

Lower Bound (of a Block)--numerically, the smallest address of a Block.

Memory Block--a Block composed entirely of either RAM or ROM.

Memory Pointer (MP)--a pointer whose value equals the address of a memory cell or port location to be read or written as data.

Memory Read (MR) Control Line--a control line indicating, when asserted, that the current access is an ADM for reading.

Memory Write (MW) Control Line--a control line indicating, when asserted, that the current access is an ADM for writing.

Object--a Block that is accessed by a Subject.

Object Index--a unique index corresponding to the Object Block pointed to by the current value of either IP or MP. The Object Index is used as a column coordinate for locating the current Rights Vector.

P--the number of Process Blocks in the hypothetical system described in the conceptual design example in this report.

Port Access (PA) Control Line--a control line indicating, when asserted, that the current access is to a Port Block.

Port Block--a Block composed of contiguous port addresses.

Process Block--a Memory Block that is executable. This implies that the Instruction Pointer (IP) can point to addresses within a Process Block. A Process Block could potentially also be read or written as though it were data.

Process Timer--a timer used to measure how long a process has been executing.

R--(from the conceptual example) the "read" rights bit of the Rights Vector; a value of one indicates that the Subject is allowed to read the Object Block to which this Rights Vector corresponds.

Reading--a Subject is said to be reading from an Object when the Subject executes an instruction that causes the Memory Pointer to point to an address that falls within (and inclusive of) the lower and upper address bounds of the Object and the MR (Memory Read) signal line is asserted.

Rights Vector--a digital pattern that contains information about what rights a Subject (Process) Block has with respect to an Object Block. A Subject will be assigned a Rights Vector for each Block in the system (including one for itself). A complete set of Rights Vectors, therefore, comprises a matrix of vector values.

Subject--the Process Block that is currently executing.

Subject Index--a unique index corresponding to the Process that is currently executing (the Subject). The Subject Index is used as a row coordinate for locating the applicable Rights Vector.

Trusted Control Lines--signal lines derived from control bits (C0, C1, and C2 in the conceptual example) contained in the Rights Vector of a Subject-Object pair. These lines can be used to enable ports or to control critical system resources.

U--(from the conceptual example) the "used" bit of the Rights Vector; a value of one indicates that this Rights Vector is in use and is, therefore, valid.

Unused Block--a Block that is not to be used as a Process Block, a Data Block, or a Port Block. An Unused Block need not correspond to addresses of actual physical memory elements contained in the system.

Upper Bound (of a Block)--numerically, the largest address of a Block.

W--(from the conceptual example) the "write" rights bit of the Rights Vector; a value of one indicates that the Subject is allowed to write the Object Block to which this Rights Vector corresponds.

Writing--a Subject is said to be writing to an Object when the Subject executes an instruction that causes the Memory Pointer to point to an address that falls within (and inclusive of) the lower and upper address bounds of the Object and the MW (Memory Write) signal line is asserted.

X--(from the conceptual example) the "execute" rights bit of the Rights Vector; a value of one indicates that the Subject is allowed to execute (transfer control to) the Object Block to which the Rights Vector corresponds.

Introduction

In microprocessor-based systems, the microprocessor usually has a variety of duties to perform. There is usually a single processor. That processor is normally capable of addressing any portion of memory (RAM or ROM) or communicating with any peripheral device. When and in what way the processor accesses each resource (RAM, peripherals, etc.) is controlled by instructions stored in either RAM or ROM. Many of these systems have little or no designed-in mechanism for checking, in real time, that the processor is performing as it should. For a system like this, one simply observes the system's functional outputs; when these outputs appear to be incorrect, he deduces that the system has malfunctioned. Some single-processor systems employ software-based checking that attempts to detect improper system behavior. This is better than no checking, but this approach cannot be said to provide independent verification that the system is functioning properly, because the processor is policing itself and other system components. Finally, some systems have special hardware components which, in combination with software measures, are used to detect error conditions. This last approach to error detection is the most effective; however, it is also the most costly.

The first approach (i.e., performing no checks) is probably not acceptable in any system. Software-only checking is sufficient in many noncritical applications where the penalty of a malfunction is not high. But in cases for which a system error can lead to highly undesirable consequences (in terms of expense or damage to property or human life), perhaps only the last approach is acceptable. Several design approaches exist for software-only checking, and there are many ways to implement checking with hardware support. But before deciding upon a particular approach, the designer should consider what he wants the checking to accomplish; that is, what security policies should the policing mechanism enforce?

In this report, we present a set of security policies for a microprocessor-based system. This is not the only possible set of policies one could devise, but this set is easy to understand, and it could be implemented, given that certain design costs and trade-offs could be accepted. The initial discussion will deal only with the policies themselves, independent of their implementation. Later, we will discuss how these policies can be implemented by describing a specific implementation of an independent hardware-based monitor.

Scope and Intent

We intend these concepts to apply to small-scale embedded systems up through dedicated (i.e., single-purpose) PC-based systems. We also assume that, if these concepts are applied to something as complex as a PC, that system supports only a single, custom-designed application program.

We intend to apply these security policy concepts to embedded-processor applications in the areas of access-control systems (where security is a prime consideration) and satellite systems (where high reliability and fault tolerance are required).

Assumptions

Several assumptions will be made in the discussion of the security policies and the implementation example. First, we assume that the system-specific parameters used to describe the security policies will be fixed at the time the system begins executing and will not change with time. That is, the policies are static in nature. We also assume that the designer can define these parameters in absolute terms with respect to memory address bounds and port addresses of the system. We assume that all executable instructions come from either RAM or ROM memory. Finally, we assume that all real-time information required to enforce the policy is available to the mechanism policing the system. Specifically, the approach we propose would not be practical for a system using a processor that has all RAM and ROM within the processor chip. We will later discuss how some restrictions mentioned here could be lifted.

Classes of Errors and Their Sources

Error Types

You will see later that the kinds of errors our security policies are effective against are those that either immediately, or after some period of propagation, cause the processor to try to access memory or peripherals in some way that the policy considers to be improper or illegal. Therefore, we will be able to detect many errors that affect either the transfer of control (via the Instruction Pointer) within the program or the addressing of data memory or peripherals (via the Memory Pointer). Of course these are not the only types of errors that can occur. For example, it would be possible for a Process to make a number of erroneous computations, return the incorrect results to memory as stored quantities, and never have its control flow or memory addressing behavior corrupted by the errors. The policies we propose here would not detect errors of this type.

Error Sources

We don't want to single out any particular source of error; rather, we'll just mention some possible ones. Errors can either be hardware- or software-based. If the error is of hardware origin, it may have been caused by a design error or by a component malfunction such as a defective memory, a bad peripheral, a faulty interface or glue circuit, or by a defect in the microprocessor itself. A hardware malfunction could be permanent (a damaged gate) or it could be transient (this could occur in a radiation environment).

Software errors will originate in one of the following ways. The software specification might be either incomplete or incorrect. The high-level software design may not meet the specifications. Finally, the programmer may not accurately implement the design. In most cases, software errors will be unintentional; however, in some cases, errors might be intentionally included. Any of the sources discussed here could cause errors that would manifest themselves in corruption of either control-flow or memory-access behavior.

Motivation for the Approach

Many of the concepts presented here are adaptations of ideas taken from the open literature on computer security. Over the past twenty years, a significant amount of research and development work has been done relating to secure computer installations and secure operating systems. A search of computer security literature will yield a majority of references dealing with mainframe computers, not microprocessor-based systems. While PC security issues have recently become a popular topic, most articles on computer security before 1984 were concerned with machines no smaller than a mini-computer. Security was discussed in the context of a large machine running multiple applications programs, either on a batch or time-sharing basis. References [1], [2], [3] and [4] represent these types of papers. In these articles, the concept of security policies is introduced. Executing programs are viewed as processes or subjects; other programs, data structures, or peripherals, are viewed as objects. A discretionary policy allows a subject to define the access rights of other subjects with respect to any object it creates. A mandatory policy places each subject or object into one of a number of classification categories (for example, unclassified, confidential, and secret). Composite security policies apply the discretionary and mandatory policies simultaneously while adding further constraints, such as the rule that no subject can write an object to a lower classification level. These policies are intended to help protect sensitive information and prevent damage and misuse of objects by unauthorized subjects.

Not all of the concepts described in the referenced papers can be applied directly to the problem of error detection on a single-microprocessor-based system. However, we believe that the idea of deriving and enforcing security policies is one that can be applied successfully to any system, regardless of its size. These policies give the designer a formal mechanism for defining, in a macroscopic way, how the system should behave.

A First-Order Security Policy

For an embedded, one-processor system, the most basic, or first-order, policy would be to divide all available RAM and ROM memory into conceptual segments, as shown in Figure 1. This segmentation is not meant to imply that all portions of segments of a particular type (DATA RAM, UNUSED ROM, etc.) are contiguous.

Figure 1. Memory Segmentation; First-Order Policy

<u>RAM</u>	<u>ROM</u>
EXECUTABLE	EXECUTABLE
DATA	DATA
UNUSED	UNUSED

The first-order policy is then defined by Table 1, below. In Table 1, a YES entry indicates that the mechanism enforcing the policy will allow the action indicated in the column heading. A NO entry indicates that the enforcement mechanism will not allow the action indicated. For example, if the microprocessor tried to execute from an area of RAM to be used only for data, the mechanism enforcing the policy would detect the error and disallow the illegal action. NOTE 1 refers to the fact that one could forbid the reading of executable RAM or ROM, if desired. If these executable memory areas were to be checked for integrity (e.g. through use of checksums), reading of these areas would have to be allowed.

Table 1. First-Order Security Policy

<u>MEMORY CLASS</u>	<u>EXECUTE</u>	<u>READ</u>	<u>WRITE</u>
EXECUTABLE RAM	YES	YES (NOTE 1)	YES
DATA RAM	NO	YES	YES
UNUSED RAM	NO	NO	NO
EXECUTABLE ROM	YES	YES (NOTE 1)	NO
DATA ROM	NO	YES	NO
UNUSED ROM	NO	NO	NO

Notice that we have assumed nothing about the inner workings of the executable code. We have simply segmented memory into six different areas and created a set of rules that states how these areas can be used by the microprocessor (e.g., READ, WRITE, etc.). We also haven't considered peripherals yet; however, peripherals could be treated much like RAM memory, with the exception that peripherals would not be executable. In other words, instructions would not be fetched directly from peripherals by the microprocessor and immediately executed.

A Second-Order Security Policy

Segmentation of Resources

Now we will further subdivide RAM and ROM and include peripherals to define a more detailed security policy. We will divide all available RAM, ROM, and peripheral-port addresses into a number of Blocks (see Definitions). Blocks consist of groups of contiguous addresses, and Blocks don't

overlap one another (no common addresses), but they completely cover all memory and peripheral (Port) resources (every available address is in some Block). The diagram below depicts this segmenting of the resources.

Figure 2. Resource Segmentation; Second-Order Policy

<u>RAM</u>	<u>ROM</u>	<u>PORTS</u>
RAM BLOCK 1	ROM BLOCK 1	PORT BLOCK 1
RAM BLOCK 2	ROM BLOCK 2	PORT BLOCK 2
RAM BLOCK 3	ROM BLOCK 3	PORT BLOCK 3
:	:	:
:	:	:
RAM BLOCK N	ROM BLOCK M	PORT BLOCK K

Our security model admits several different types of Blocks: Process Blocks, Data Blocks, Port Blocks, and Unused Blocks (see Definitions). A Process Block contains executable instructions and may contain some data. A Data Block contains only data and is not executable. Port Blocks will generally be treated like RAM Data Blocks, except in the case of direct memory access (DMA), when a Port, in cooperation with a DMA controller, acts like a Process. We will treat DMA later when we discuss extensions to the basic second-order policy.

Executing, Reading, and Writing

The Definitions section gives exact meanings of "Executing", "Reading", "Writing", "Instruction Pointer" (IP), and "Memory Pointer" (MP) within the context of this report. That section also defines certain control lines-- "Instruction Fetch" (IF), "Memory Read" (MR), and "Memory Write" (MW). These latter control lines can be used to determine whether the address currently on the bus is the IP or the MP (see below).

Table 2. Bus Content Versus Control Lines

<u>Control Line States</u>			<u>Quantity on Address Bus</u>
<u>IF</u>	<u>MR</u>	<u>MW</u>	
1	0	0	IP
0	1	0	MP
0	0	1	MP

The question of whether MP is pointing at memory (RAM or ROM) or a port is resolved by use of a fourth control line, "Port Access" (PA). We defer further discussion of PA until later.

Execution of a Block is considered to be occurring whenever the IP points to an address that is within that Block. Read or write operations on a Block are occurring whenever the MP points to an address that is within that Block. It is apparent that what we are calling a Process Block may, in fact, be what is commonly thought of as a "procedure", "subroutine", or "function"; and what we are calling a Data Block may be composed of one or more "data structures."

The Concept of Access Rights

An executing Process can be thought of as a Subject (see Definitions) in the sense that it has control of the processor. Also, all Blocks defined for the system can be referred to as Objects because they may potentially be used in some way by the Subject. The Subject may execute another Process Block by causing the IP to point to an address within the bounds of the Object Process, or the Subject may read from or write to an Object Block.

Usually software can be designed so the number of Object Processes a Subject needs to execute is somewhat limited. Similarly, a Subject usually needs to read or write only a limited subset of the data memory. In assigning access rights to a Subject, we explicitly declare which Objects are accessible to that Subject and define exactly how the Subject can access each Object.

The Second-Order Security Policy model described here assigns to each Subject a set of Rights Vectors (the format of these Vectors will be discussed later in a specific example), one for each Block in the system. Each Rights Vector is a small number of bits (one to two bytes) containing the rights of the Subject with respect to the Object Block. Each Subject/Object pair is assigned its own Rights Vector. So, for a system partitioned into a number (B) of Blocks of which a number (P) are Processes, we would need to define a matrix of B-times-P individual Rights Vectors to specify the security policy for the system. Using the Rights Vectors, we can, for example, declare that Process A can execute Processes B and C, but none other (Figure 3), and that Process A can read and write Blocks X and Y, but none other (Figure 4).

Figure 3. Rights to Execute an Object

(SUBJECT)		(BLOCKS)	POLICY RULING
PROCESS A	----->	PROCESS B	--- PERMITTED
ATTEMPTS TO EXECUTE	----->	PROCESS C	--- PERMITTED
	----->	PROCESS Q	--- ** ERROR, ** ** DENIED **

Figure 4. Rights to Read Or Write an Object

(SUBJECT)		(BLOCKS)	POLICY RULING
PROCESS A	----->	BLOCK X	--- PERMITTED
ATTEMPTS TO READ FROM OR WRITE TO	----->	BLOCK Y	--- PERMITTED
	----->	BLOCK Z	--- ** ERROR, ** ** DENIED **

A Priori Rights Possible

As with the First-Order Policy, we can deduce something about the content of some of the Rights Vectors even without knowing anything about a particular system to which these concepts might be applied. The following matrix describes what we can say, a priori, about the rights of the Subject with respect to an Object of a particular type.

Table 3. Second-Order Policy: A Priori Rights Possible

<u>OBJECT TYPE</u>	<u>MODE OF ACCESS TO OBJECT</u>		
	<u>EXECUTE</u>	<u>READ</u>	<u>WRITE</u>
RAM PROCESS	Y/N	Y/N	Y/N
ROM PROCESS	Y/N	Y/N	N
RAM DATA	N	Y/N	Y/N
ROM DATA	N	Y/N	N
PORT	N	Y/N	Y/N
UNUSED	N	N	N

Entries having a single letter are rights whose value can be specified a priori; that is, without any knowledge of the application. "Y" means the access is allowed; "N" means the access is not allowed. An entry of "Y/N" means that this type of access may or may not be allowed at the discretion of the designer.

Ideas for Partitioning the System

Why Partition? The designer may want to partition his system into Blocks and devise a way of enforcing security policies for the sole purpose of detecting as many errors as possible. However, the desire to partition can also arise when some aspects of the system are more critical than others. A function of the system may be critical--perhaps the designer wants to assure that a particular group of peripherals (a Port Block) is activated only by a specified section of executable code (a Process Block). Similarly, certain data structures of the system may be very important--the designer might want to make sure that only one or a few Processes have access to these structures. Finally, the designer may want to limit the ways one Process can access another--for example, Process A might be able to execute Process B, but the designer could assure that A could neither read nor write B as data (such writing would constitute self-modifying code, an undesirable practice).

Granularity. Any degree of granularity can be used in partitioning the system. The designer could define only a few Blocks, tens of Blocks, or even hundreds. In one case a Process could be equivalent to a procedure (a subroutine or function). In another, a group of procedures could be considered to be a Process (coarser granularity). However, extremely fine granularity could be achieved by defining one very important machine-language instruction to be a Process. Similarly, the granularity of Data Blocks can be as fine as one byte and as coarse as an arbitrarily large number of bytes. Of course, the system can be segmented into a mixture of small, medium and large-sized Blocks, because a Block's required size is determined by the needs of the system, not by the sizes of the other Blocks. In practice, granularity will be determined by a trade-off between implementation costs (dollars, volume, and complexity) and the desired security level of the system.

Assigning Rights. Once the system Blocks are identified, the Rights Vector for each Subject-Object pair should be given separate consideration, and its value should be defined completely. In each case under consideration, rights to access an Object in a particular way should be granted to a Subject Process only if that Subject needs that access to do its job.

Implementation Methods Possible

Several implementation schemes could be used to enforce the security policies described in this report. The three approaches discussed here are the following--

1. enforcement done totally in software (no hardware support);
2. enforcement implemented using designed-in, internal security features of the microprocessor around which the system has been built;
3. enforcement accomplished by adding an external monitor to the basic system; the monitor can be either processor-driven or composed of custom-designed logic.

The following discussion compares each method of enforcement to the case of a system employing no enforcement mechanism.

Software-Only Enforcement

Enforcement totally in software might be implemented using an artificially constructed instruction set. Each artificial instruction would be either a macro or a subroutine call. The source code would consist of statements composed of only these "allowed" artificial instructions plus a selected subset of the native instruction set of the host microprocessor. Each artificial instruction would translate to assembly code capable of performing the additional checking required to assure that any control transfer or memory access conforms to the security policy. This approach would result in a much larger body of software than that for a system with no enforcement mechanism. Execution speed would be retarded significantly; however, system function would otherwise be the same. Details of implementing the policy enforcement would strongly depend on the system application, and use of software enforcement would, by its nature, place significant constraints on the software design. Software development time would be increased; however, the overall reliability impact would probably be positive. The fielded application would probably have fewer errors, and the design would be easy to change. Inherent security of the system would be moderately improved. However, because the security software would run on the one and only processor in the system, such software could not be considered to be an independent positive measure.

Use of Internal Security Features of the Processor

Through use of internal security features of a microprocessor, one might construct a security kernel that could mediate requests, by each Process, for access to the various resources (RAM, ROM, Ports) of the system. The Intel 80286 [5] is an example of a microprocessor with these kinds of internal features. The software required to support this approach would be somewhat larger than for a system with no enforcement, and as with software-only enforcement, execution would be much slower. Otherwise, system function would be the same. Use of this approach would have a significant impact on the design and implementation of the software; however, the software techniques used, and certainly the hardware mechanisms (which are internal to the processor) would tend to be reusable from one application to the next (not strongly application-specific). Comments made for software-only enforcement relating to development time, reliability, and number of fielded errors apply here as well. Significant improvement would be realized in inherent system security, and this approach gives some measure of independence (from the executing software) in detection of security violations.

External Hardware-Based Monitor

One might construct an external hardware-based monitor to enforce the security policies. If the monitor were processor-driven, system operating speed would be retarded. A nonprocessor implementation that would allow the system to run at full speed is preferable and is assumed here. No additional software would be required to support the monitor. System function would be the same as for a system with no enforcement mechanism. Although size of the software would not be affected, the control flow, module structures, and data structures would need to be tailored, in some respects, to conform to monitor requirements. The monitor would be reusable on other system applications having the same processor. Development time would be increased. Costs would be incurred in the monitor design and in the extra volume the hardware would occupy in the fielded application. This approach promises a reduction in the number of fielded software errors. Inherent system security is greatly enhanced, and this approach provides a positive measure that can be totally independent of the host microprocessor and its software.

A Conceptual Design Example

We present here a hypothetical system and describe how a hardware-based Monitor could be constructed to enforce the second-order security policy described earlier. We assume that the totality of system resources has been logically partitioned into a number (B) of Blocks (Process, Data, and Port), of which a number (P) are Processes.

System Description

We will not specify the function of the system; however, as depicted in Figure 5, the system contains RAM and ROM and can communicate to peripherals through a number of I/O ports. Figure 5 also shows the Hardware Monitor (to be described below). Note that the Monitor has access only to the processor's control lines and the address bus and not to the data bus.

Outputs of the Monitor include an "error" line, which indicates when an error has occurred; an "error register", which contains information about the type of error; and "trusted control lines", to be described later. The error line is used here as a hardware interrupt to the processor. In this implementation, an interrupt handler evaluates the contents of the error register and decides upon a course of action--attempted recovery, system reset, or system shutdown. In the case of attempted recovery, the interrupt handler can reset the Monitor's error register, allowing processing to proceed.

The error line could also be processed in hardware and not fed back to the processor. This might be done if no recovery is attempted and the system is simply to be either reset or shut down in response to an error condition. In this case, the system reset would be used to reset the Monitor's error register.

The Hardware Monitor

Figure 6 shows a diagram of the three Hardware Monitor components. The Block Discriminator Bank determines the index of the Object Block being accessed by the processor. Having kept track of which Process is the Subject, the Rights Vector Decoder uses the Object Index to decode the Rights Vector for the current Subject/Object pair (coincidence). The Rights Vector is then presented to the Authenticator, which determines whether an error has occurred and, if so, the type of error.

Figure 5. Hypothetical System with Monitor

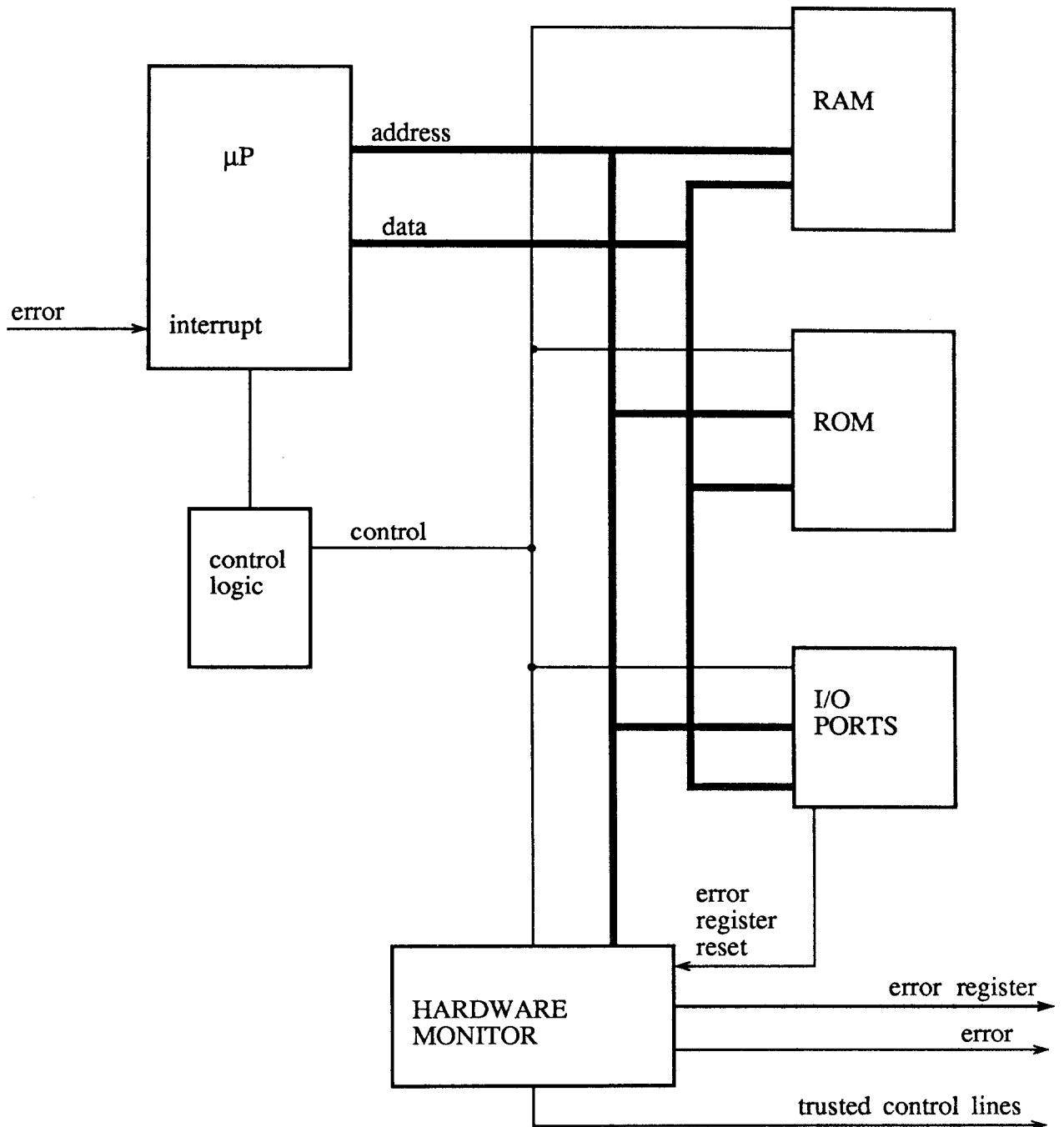
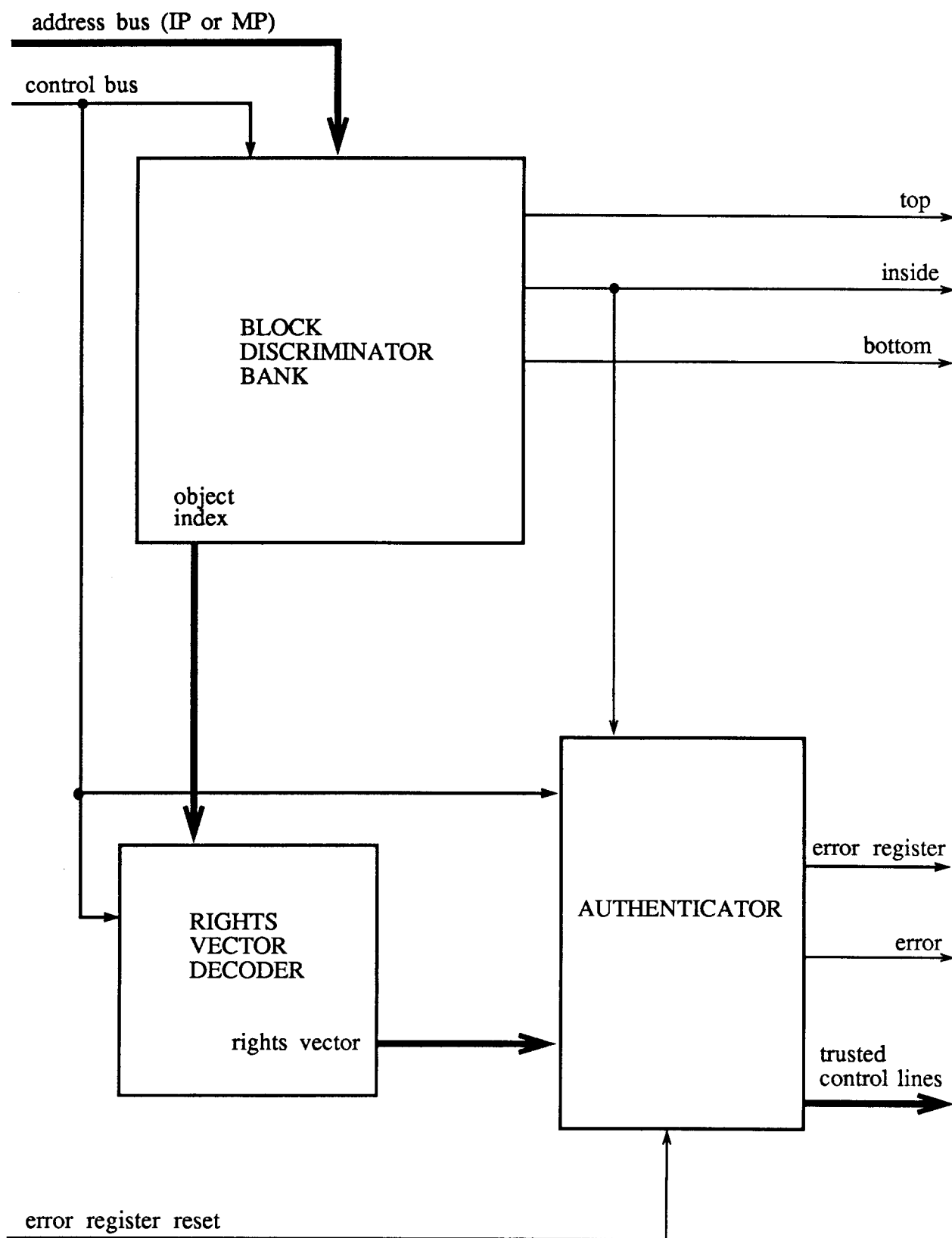


Figure 6. External Hardware Monitor



Block Discriminator Bank. By decoding the address bus and the system's control lines, the Block Discriminator Bank determines which Block in the system is being accessed on the current Access to Instruction Memory (AIM) or Access to Data Memory (ADM) (see Definitions). Figure 7 shows details of the Block Discriminator Bank. One output of the Block Discriminator Bank is the Object Index; this index identifies the Block being accessed by the Subject Process. The Object Index is Q bits long; this is the number of bits required to express the number (B) of system Blocks. The other outputs are the lines "inside", "top", and "bottom." The inside line is asserted whenever the address-bus contents points to a location within one of the B Blocks of the system. Failure of this line to be asserted on either an AIM or ADM would constitute an error condition. The top and bottom lines are asserted whenever the address-bus contents point to either the first or last address, respectively, of the Object Block. See Entry and Exit Point Enforcement, page 37, for a discussion of the possible uses of "top" and "bottom."

Block Discriminator Cell. The primitive discrimination element is the Cell, shown in Figure 8. Each Cell contains a stored copy of the first address (lower bound) and last address (upper bound) of the Block to which it corresponds. A window detector determines whether or not the current address on the bus falls within the bounds of the Cell. Ambiguities between memory and port addresses are resolved using the Port Access (PA) line, not previously mentioned in the description of the Block Discriminator Bank. Although Figure 8 shows a "Pointer Register", no latching of the pointer is required at this point, avoiding the need for any clocking of the Block Discriminator Bank.

Rights Vector Decoder. The Rights Vector Decoder is shown in Figure 9. Control lines and the Object Index serve as inputs. The Object Index is latched into the Object Index Register by either IF, MR, or MW. Recall that the Object Index identifies the Block being accessed by the Subject on the current AIM or ADM. The Subject Index for the current decode will have been latched into its register on the last previous AIM. The Subject-Index/Object-Index pair provides a row-column address into a ROM that contains the Rights Vector information. Once the Rights Vector is stable, it can be clocked into its register on the falling edge of either IF, MR, or MW. The Subject Index will be updated only if the current access is to instruction memory (i.e., an AIM). Therefore, the update can

Figure 7: Block Discriminator Bank (B Cells)

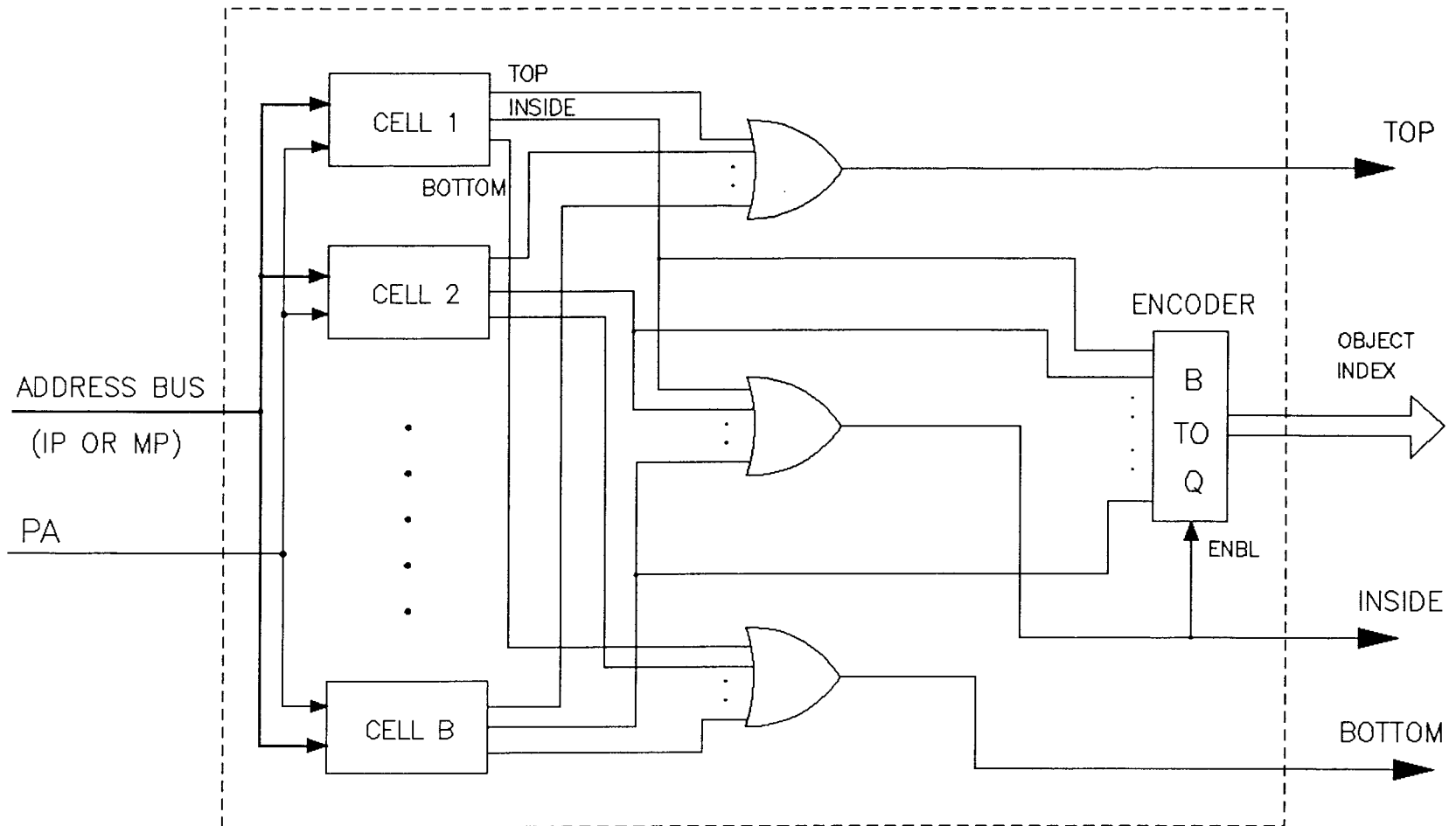


Figure 8. Block Discriminator Cell

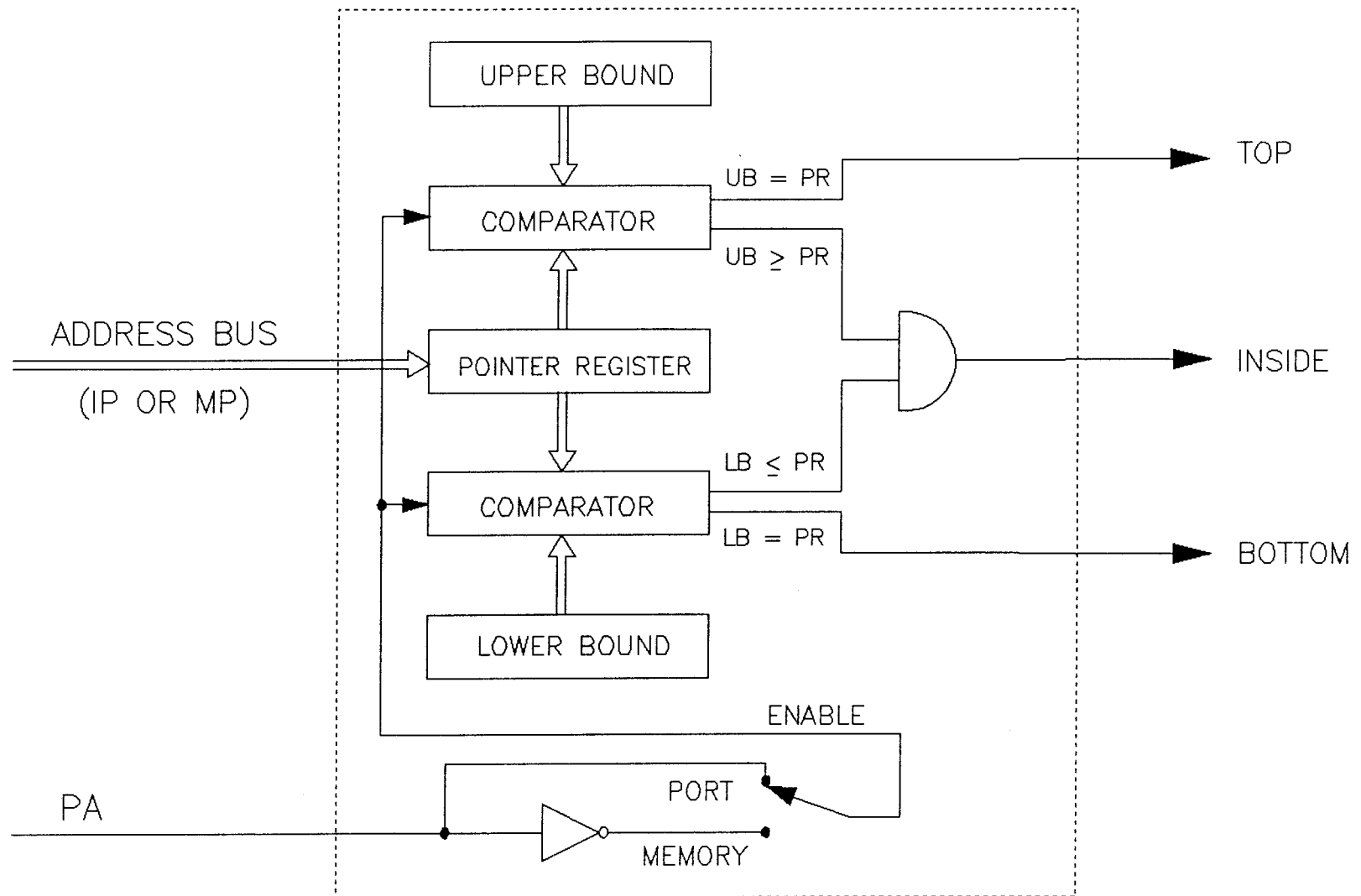
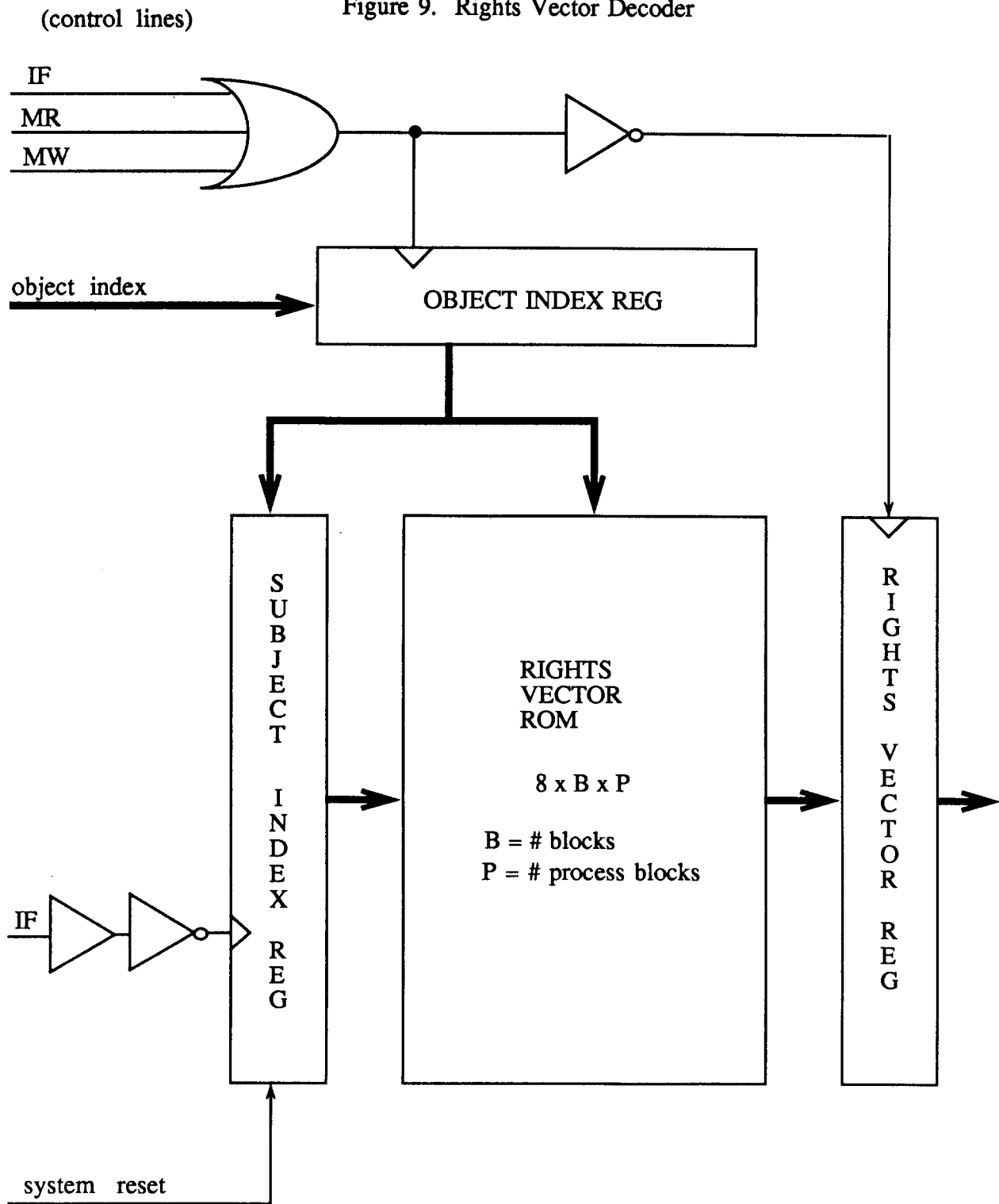


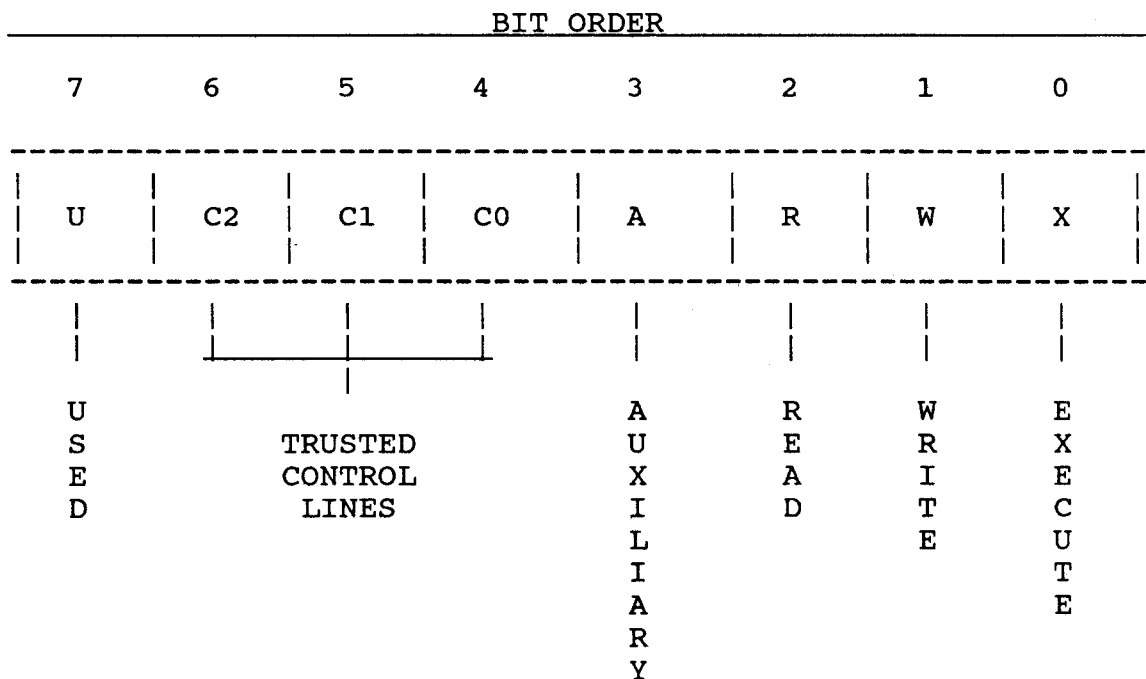
Figure 9. Rights Vector Decoder



be clocked by a slightly delayed falling edge of the IF control line. The "system reset" line (herein assumed to be one of the system control lines) initializes the Subject Index to the index of the bootstrap Process Block when the system is started or reset.

Rights Vector Format. The format of the Rights Vector is shown in Figure 10, below.

Figure 10. Rights Vector Format



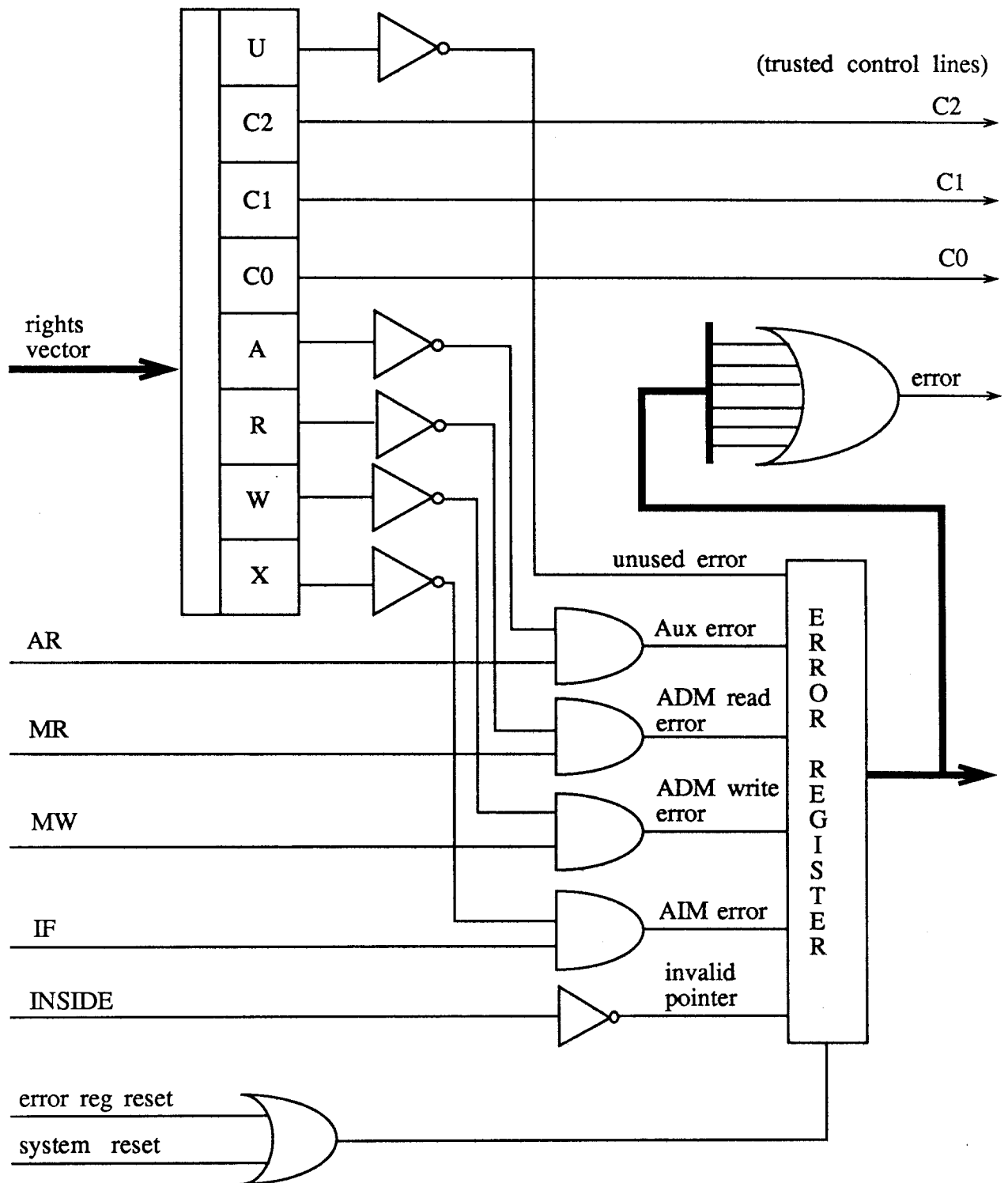
The U bit is one when the corresponding Object Block is valid and in use. R, W, and X indicate, when asserted, the right of the Subject to read, write, or execute the Object. The A bit, when asserted, indicates that the auxiliary operation indicated by the Auxiliary Request (AR) line is allowed for the current Subject/Object coincidence. Use of this feature allows the Monitor to enforce known relationships between the status of the auxiliary operation (via the AR line) and the current state of the system (the Subject/Object pair). The Trusted Control Lines contain a code available to the system.

This code can be used with the assurance that the system state corresponds to one or a small number of known Subject/Object coincidences. These lines can be used to enable or inhibit data gates or functional control lines based upon this knowledge of the system state. The three trusted control lines we show can represent a total of eight system states. More control lines could be added to the Rights Vector to increase the number of represented system states. Representation of all possible Subject/Object coincidences would require a number of bits equal to the base-two logarithm of B times P. If this many states were to be represented, this information would not be included in the Rights Vectors. Rather, the Subject Index and Object Index would be made available to the system as outputs of the Rights Vector Decoder. This would allow the system logic to directly decode and act upon this detailed and trusted knowledge of the system state. Most likely, a smaller number of trusted control line bits would be needed in practice. Our example provides for only three--C2, C1, and C0.

The Authenticator. Figure 11 shows a block diagram of the Authenticator. The Rights Vector, system control lines, the inside line, and the "error register reset" line serve as inputs. If U is zero, the Subject is trying to access an unused Object Block, and an error condition is indicated. If R is zero (reading not allowed) and MR is asserted (reading is attempted), an "ADM read error" is detected. If W is zero (writing not allowed) and MW is asserted (writing is attempted), the "ADM write error" line is asserted. Attempting to execute (IF asserted) when X is zero (execution not allowed) will cause "AIM error" to go to one. An attempted auxiliary operation (AR asserted) without a value of one in A will cause "Aux error" to go high. Finally, any access to resources for which the "inside" line is not asserted would cause the "invalid pointer" line to go to one.

Error signal lines are collected in the error register. All error lines are or-gated together to create the error output line. The error register will be reset by the error register reset, which comes from the processor through an I/O port, or by the system reset. The trusted control lines come directly from the Rights Vector register to the output of the Authenticator. These lines would be decoded or used as discrete line inputs to the system logic or, possibly, to the processor via an I/O port. The diagram shown in Figure 11 is conceptual in that required latching for control lines MR, MW, IF, etc., and for the error register is not shown.

Figure 11. Authenticator



Handling of Interrupts. Interrupts can be accommodated with the Monitor. Interrupt service routines will be treated just like any other Process and assigned to Process Blocks. All Processes in which interrupts are enabled will be given "read" access to the Data Block(s) where the interrupt vectors are stored. If a Process can be interrupted, it must have execute rights to the handler of the corresponding interrupt. Generally, a Process should not be granted execute rights to a handler of an interrupt that is disabled during the time the Process executes. The need to define which handlers each Process has rights to will force the designer to explicitly define a philosophy or policy for handling interrupts.

Once a handler starts executing, it must be treated just like any other Process, so the designer must define the rights of this handler to all other Blocks, in particular, to handlers of subsequent interrupts that might occur. This again points to the need for a well defined plan for enabling and prioritizing interrupts.

The Work of Namjoo and McCluskey

We discovered References [8] and [9] after we had completed the first draft of this report. These references reveal that Namjoo and McCluskey originated many of the concepts we describe in this report concerning verification of proper memory-access behavior using an external watchdog device. Here we contrast and compare their work with ours. They propose using a processor-based external monitor; we propose a hardware implementation. Their approach appears to be targeting larger systems, i.e., machines in the mini-computer class with a complex operating system (the VAX-11 is used as an example). While our hardware approach could also be implemented in larger systems, the scope of our work is limited to small, microprocessor-based systems that may not have an operating system. They propose dynamically changing the partitioning and access-rights information of the system so that different software applications can run under different sets of security parameters. Trusted software, therefore, must load these parameters. Our approach, which assumes that a single application is executing, uses unchanging security parameters. Their "Segment Mapping Table" defines how resources are partitioned. The "Segment Access Table" corresponds closely to our Rights Vector matrix. The Rx and Ry registers of [8] correspond to our Object Index and Subject Index, respectively. The work of Namjoo and McCluskey predates ours, and there is a significant amount of overlap between their results and ours. Nonetheless, we feel that our more detailed treatment of the security policies and our conceptual design of the Hardware Monitor serve to augment and extend the results they have already reported.

Interpretation of the Approach's Significance

This section interprets the significance of the security policies and of their enforcement using an independent Hardware Monitor.

A Positive Security Measure

Effective enforcement of the second-order policy provides a positive security measure against erroneous behavior of a microprocessor-based system. The external monitor approach has several advantages:

- the positive measure is easily identifiable; that is, one can point to it;
- the policies and the Monitor design are very simple; therefore, they can be understood and analyzed;
- the Monitor is a separate entity; so, it is independent of the microprocessor and its executing software.

Functional Advantages

- the Monitor can run in real time without slowing down the processor;
- the Monitor's design should be reusable on different systems that use the same microprocessor;
- major portions of the Monitor could be implemented in a custom VLSI circuit to minimize its size.

Errors the Monitor Can Detect

The Monitor can detect errors that directly or indirectly affect either the Instruction Pointer (IP) or the Memory Pointer (MP). Some of these types of errors are--

- corruption of the stack causing invalid addresses for data accesses or control transfers;
- corruption of the IP causing an invalid instruction fetch;

Errors the Monitor Can Detect, continued...

- illegal data accesses caused by hardware errors or coding errors, especially in code with complex data structures for which addresses are computed, e.g., linked lists;
- illegal control transfers caused by hardware errors or coding errors, especially in code with computed control transfer destinations;
- invalid interrupts or interrupts occurring with no interrupt handler.

Damage Confinement

Use of the external Hardware Monitor will reduce the amount of damage to RAM data and instruction areas and to attached peripheral devices that can occur when memory access errors go undetected. In a partitioned, monitored system like the one we have defined in our conceptual design example, direct damage caused by erroneous behavior of a process is confined almost exclusively to Blocks to which the Process has been assigned rights of access. The notable exception is that an erroneous write access probably cannot be prevented even though the Monitor detects the error before the end of the bus cycle on which the illegal write occurs.

A Basis for a Fault-Tolerant System

Given the atomic structure of the system (i.e., the separation and independence of the Blocks), this error-detection scheme could provide a basis for implementing a fault-tolerant system that is able to recover from errors [6,7]. Such a fault-tolerant system would save its state whenever control passes from one Subject to the next. The state would be just the Object Blocks the current Subject has "write" access to, in addition to the state of the processor itself.

Another Philosophy of Use and Benefits

Even if the system does not attempt recovery when an error occurs, significant benefits accrue from this approach. Foremost, the partitioning and its enforcement can guarantee certain conditions that provide the designer with a true security enhancement. This will remove from the software some of the burden of providing security. More importantly, it will make the system much easier to analyze from a security

point of view. For example, if a particular port is attached to a sensitive or valuable resource, and we want to assure that this port is only addressed from a particular Process, we can enforce this behavior through use of the Rights Vectors and the external Monitor. Any security analysis of the software can then concentrate on only that portion of the software that can legally access the critical port.

Reduction in the Number of Fielded Errors

The rigid definition of the access rights of each Subject to each Object and the integration of the Monitor into the system during the development process will likely allow detection and elimination of design errors that would not have been detected without the Monitor.

Limitations of the Approach

Certainly, error propagation from one Process to another is still possible without immediate detection. For example, a process might write erroneous data to memory without making any accesses that the Monitor would declare illegal. Any other Processes having read-access to this memory could suffer errors from the contaminated data. Likewise, a Process could pass control to another Process (to which it has rights of access) in an erroneous way. The erroneous transfer could occur at the wrong time; or the transfer destination, though within a proper executable Block, may be incorrect.

These second-order error-propagation mechanisms can be countered, in many cases, by using dynamic software-based error checking. For example, redundancy checks of stored data and reasonableness checks of results of computations can improve integrity of the stored data. Likewise, the program can dynamically check its own control flow by recording a history of the path of the instruction pointer and comparing this history to a stored description of the program's expected behavior. Therefore, an integrated approach, using the external Hardware Monitor along with dynamic software error checking, can yield broader error coverage than the Monitor alone can provide.

Extensions to the Basic Approach

We believe that the Hardware Monitor we have described can be built and will be effective. We intentionally avoided adding more complicated features to the policies and the Monitor so the presentation would be clear and simple. However, features described below might extend the capabilities of the Monitor and make it more effective. Of course, the added features would increase the Monitor's complexity.

A Watchdog Process Timer

A watchdog timer might be incorporated into the Monitor to measure the execution time of one or more software Processes. The simplest implementation would provide for a voluntary measurement, wherein a Process would load a time-limit quantity into the timer's counter and then strobe the timer, causing it to begin its countdown. The user Process would then be responsible for reading the timer and determining whether the execution time was within bounds. This approach does not possess the attribute of independence, because the Process itself initiates and evaluates the timing measurement; in other words, this approach is based upon implicitly trusting the software and the processor. It would be possible to obtain independence of the timing measurements if the Monitor contained pre-stored time-limit quantities for the Processes of interest and if the Monitor were smart enough to determine when a Process begins and when it ends. Determining when a Process begins is not difficult; the Monitor simply detects the first occurrence of the corresponding Subject Index at the output of the Block Discriminator Bank. Determining when a Process ends is difficult. The Monitor must distinguish between exits from the Process caused by "Calls" and those caused by "Returns." This will be discussed later in the section on "Entry and Exit Point Enforcement."

Ways to Reduce the Number of Block Discriminator Cells

In our example, there was one Cell for each Block of the system. A reduction in the number of required Cells would seemingly simplify the Monitor hardware and reduce Monitor cost. The following two paragraphs describe a technique that uses loadable, multi-purpose Cells to reduce the total Cell count.

Process Declares Its Index to Monitor. We could require that each Process, whenever it is entered or reentered, declare its Subject Index to the Monitor. The Monitor would then fetch the address bounds corresponding to the "claimed" Index from a ROM, transfer them to a Cell, and then verify, using a latched value of IP, that IP falls within the bounds. The Monitor could easily determine when IP exited the current Process and would expect a new Index declaration within a specified time of the exit. Failure to receive a declaration would cause an error condition. This technique, which would have little impact on processing speed and moderate impact on Monitor complexity, would reduce the number of Cells required in the Monitor hardware.

Process Declares Indices of Needed Objects to Monitor. To complete the scheme for reduction of the number of Monitor Cells, we would require that each Process, whenever it is entered or reentered, declare to the Monitor the indices of all Objects to which it requires access. The Monitor would then fetch the address bounds of each of the "requested" Objects from a ROM and transfer each to a Cell in the Block Discriminator Bank. This technique, when coupled with self-declaration of the Subject Index described above, would reduce the number of required Cells to a number equal to one plus the maximum number of Objects required by any Process. Although the number of required Cells is reduced substantially, we have actually increased the Monitor's complexity by requiring that it lookup and load Subject and Object bounds. This approach also significantly impacts the software design. Finally, the Monitor will no longer be able to keep up with the processor.

Entry and Exit Point Enforcement

It should be possible to designate a "first entry point" and a "final exit point" for each Process and to design a Monitor that could discriminate these entries and exits from those that occur during calls-to and returns-from procedures (subservient Processes). To accomplish this, the Monitor would require access to the data bus (recall that the Monitor in the example did not require this) and would require some knowledge of the processor's control-transfer instructions (CALL, RETURN, JUMP, etc.). The "top" and "bottom" lines of the conceptual design example could also be used, in certain cases, to determine when a Process is first entered or last exited. The Monitor could use this discrimination feature to assure that the Process is always first entered and always last exited at particular known values of IP. This feature would also be useful to aid in implementing a completely independent process timer.

Enforcement of Returns to the Proper Routine

Yet another enhancement would be the ability of the Monitor to determine that a Procedure, once called, returned to the calling Process when finished. In addition to being smart enough to determine and verify "first entry points" and "final exit points", the Monitor would need a stack to store the history of the calling sequence, from Process to Process, in order to implement this feature.

Handling Direct Memory Access

Many microprocessor-based systems use DMA to speed up data-transfer times to and from peripherals. During a DMA, a smart peripheral, with the help of a DMA controller, provides addresses and data to the system buses and directly reads/writes data from/to memory or other ports, thereby bypassing the main microprocessor. The approach we would take to handling DMA is to treat the smart peripheral like a Process. Therefore, the port would be designated both as a Port Block and as a Process Block. Probably the format of the Rights Vector would not be affected by treatment of DMA. We would assume that each smart peripheral is assigned a unique DMA channel and a unique Data Block to which it will have access. We would also assume that a smart peripheral could only read or write on DMA cycles dedicated to its particular DMA channel.

Time-Varying Security Policies

We have assumed that system security parameters (Block bounds and Rights Vectors) are fixed during production of the system and do not change as the system executes. One could devise a Monitor whose security parameters could be loaded at power-up and dynamically changed at various times as the system executes. This approach is taken in [8].

Precluding Execution of Forbidden Instructions

The designer may want to assure that elements of a subset of the processor's instruction set are never executed. The Monitor could be designed to detect when any one of these forbidden instructions is fetched from instruction memory. This feature would, of course, require the Monitor to have access to the microprocessor's data bus.

Some Problems With the Approach

There are several potential difficulties with the basic approach of the second-order policy and the Hardware Monitor. This section discusses some of these problems.

Dynamic Allocation of Memory

The static partitioning required by the method does not lend itself well to dynamic (run-time) allocation of RAM. The use of dynamic memory allocation usually implies that there is an available heap of memory that can be used to provide storage areas, on demand, as needs arise.

Our method does not preclude dynamic allocation, just as it does not prevent reuse of Data Blocks for multiple purposes. However, care must be taken in controlling the management of memory because any allocation that causes an Access to Data Memory (ADM) that is not in accordance with the decoded Rights Vector will cause the Monitor to indicate an error condition. A prudent approach would be to dynamically allocate, from a large heap, memory to be used for storage of data that is considered to be in no way critical. Data Blocks to be used for critical or sensitive data should be limited to use for a single purpose; therefore, dynamic allocation is not appropriate for that type of data.

Overlays

Overlays amount to reuse of RAM by different bodies of executable code. Therefore, the problems associated with overlays are similar to those seen with dynamic allocation. The use of overlays is not precluded by the method; contents of RAM Process Blocks can change dynamically. However,

because the Rights Vectors and Process Block bounds are constant, any Processes overlaid onto a particular Process Block must fit within the bounds of that Block and must behave in accordance with the Rights Vector information stored in the Monitor.

A reasonable way of dealing with this limitation is to allow overlays for noncritical Processes and to avoid overlays for Processes performing more important or critical functions.

Use of High-Level Language

If a high-level language is used to implement the software, the designer may have difficulty maintaining absolute control over relative placement, in memory, of the various Process and Data Blocks. We do not think this is a serious problem. Nearly all compilers will preserve, or can be forced to preserve, contiguity of the compiled instructions that compose a Process Block and of data elements that compose a Data Block. Further, the linker can be made to produce a report showing the relative location and length of each Block. So, absolute control during compilation and linking is probably not necessary as long as complete knowledge of the structure of the executable file can be obtained from the linker.

Pre-fetching of Instructions

Some processors pre-fetch several instructions at one time. This creates a possible problem with respect to fetches of instructions lying near the end of a Process Block. When pre-fetching past the end of the Block occurs, the Monitor will indicate an error condition. This problem can be remedied by padding past the final exit point of the Block with null operations or some other known operational code.

Use With Commercial Operating Systems

Using the Monitor on a system employing a commercial software operating system will present several problems. The designer may not have design information and source code for the operating system. Therefore, he may not have enough information to define Block partitions within the operating system software. Even if he can do the partitioning, he cannot intelligently assign access rights if he does not understand the details of how the operating system works.

Conclusions

In developing the second-order security policy, we have borrowed security concepts normally applied to large multi-user systems and translated them into concepts that can be applied to microprocessor-based systems. We have presented a conceptual design for a Hardware Monitor that can be constructed to enforce the second-order policy in real time, without retarding the processing speed of the system. The Monitor could be reused on other systems based on the same processor.

Use of the Monitor provides an independent, real-time capability for dynamically detecting erroneous accesses to memory and to input/output ports and for flagging improper program flow from Block to Block. The Monitor can be used as an error detector to provide the basis for a fault-tolerant system. Or, the primary purpose of the Monitor can be to provide a security enhancement that removes a significant security burden from the software and greatly simplifies the task of analyzing the system from a security point of view.

The basic approach could be extended to add enhanced features, as described in a latter section of this report. With the enhancements would come increases in complexity, degradations in processing speed, and a decreased likelihood of reusability of the Monitor for different systems employing the same microprocessor. We have identified several possible implementation problems with the approach; however, none of the problems appear insurmountable.

References

- [1] Downs, Deborah D., "Operating Systems Key Security with Basic Software Mechanisms," Electronics, vol. 57, no. 5, March 8, 1984, pp. 122-127.
- [2] Popek, Gerald J., and Charles S. Kline, "Issues in Kernel Design," Proceedings of the National Computer Conference, 1978, pp. 1079-1086.
- [3] Lorin, Harold, "Emerging Security Requirements," Computer Communications, vol. 8, no. 6, December 1985, pp. 293-298.
- [4] Saltzer, J. H., and M. D. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE, vol. 63, no. 9, September 1975, pp. 1278-1308.
- [5] Wells, Paul, "On-Chip Hardware Supports Computer Security Features," Electronics, vol. 57, no. 5, March 8, 1984.
- [6] Anderson, T., and P. A. Lee, Fault Tolerance Principles and Practice, Prentice Hall, 1981.
- [7] Avizienis, Algirdas, "Fault Tolerance by Means of External Monitoring of Computer Systems," Proceedings of the National Computer Conference, 1981, pp. 6-1 through 6-14.
- [8] Namjoo, Masood, and Edward J. McCluskey, "Watchdog Processors and Capability Checking," in Digest of Papers of the 13th Annual International Symposium on Fault Tolerant Computing FTCS-12, Santa Monica, CA, June 22-24, 1982, pp. 245-248.
- [9] Mahmood, Aamer, and Edward J. McCluskey, "Concurrent Error Detection Using Watchdog Processors--A Survey," IEEE Transactions on Computers, vol. 37, no. 2, February 1988, pp. 160-174.

Distribution:

1410	P. J. Eicker
2110	R. E. Blair
2310	M. K. Parsons
2312	D. J. Allen
2312	J. D. Mangum
2315	M. J. Smartt
2330	J. H. Stichman
2331	A. E. Sill
2340	M. W. Callahan
2345	B. C. Walker
3141	S. A. Landenberger (5)
3151	W. I. Klein (3)
3154-1	C. L. Ward (8) for DOE/OSTI
5120	W. R. Reynolds
5121	D. F. McVey
5126	W. D. Chadwick
5127	J. R. Caruthers
5128	B. E. Bader
5145	J. A. Cooper
5145	D. H. Schroeder
5145	M. W. Sharp
5150	K. D. Hardin
5160	G. R. Otey
5221	D. D. Spencer
5233	D. C. Hanson
5240	D. S. Miyoshi (3)
5250	T. A. Sellers
7220	R. R. Prairie
7222	G. C. Novotny
7230	J. F. Ney
7232	T. I. Barger
7232	S. D. Spray
7233	R. M. Axline, Jr. (15)
7233	S. C. Billups
7233	B. P. Gaude
7233	J. R. Gosler
7233	M. F. Grady
7233	R. E. Smith
7234	J. M. Portlock
7251	J. M. Sjulín
8130	L. A. Hiles

Continued on following page

Distribution, continued

8134	E. L. McKelvey
8233	J. M. Harris
8524	J. A. Wackerly
9110	C. W. Childers
9210	H. M. Dumas
9211	T. G. Taylor
9212	L. Convisser
9212	G. R. Elliot
9212	J. Gomez
9212	L. W. Maschoff
9212	J. I. McNabb
9212	S. E. Murray
9212	R. C. Ormesher (15)
9212	S. A. Spraggins
9212	J. W. Zipay
9220	G. H. Mauth
9230	R. E. Spalding
9231	J. C. Chavez
9231	E. P. Evans
9231	D. J. Kral